Memoization

From Wikipedia, the free encyclopedia

In computing, **memoization** or **memoisation** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization has also been used in other contexts (and for purposes other than speed gains), such as in simple mutually recursive descent parsing[1] in a general top-down parsing algorithm[2][3] that accommodates ambiguity and left recursion in polynomial time and space. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, memoization is also known as tabling;[4] see also lookup table.

Contents

Etymology

The term "memoization" was coined by Donald Michie in 1968[5] and is derived from the Latin word "memorandum" ("to be remembered"), usually truncated as "memo" in the English language, and thus carries the meaning of "turning [the results of] a function into something to be remembered." While "memoization" might be confused with "memorization" (because they are etymological cognates), "memoization" has a specialized meaning in computing.

Look up ***memoization*** in Wiktionary, the free dictionary.

Overview

A memoized function "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it, thus eliminating the primary cost of a call with given parameters from all but the first call made to the function with those parameters. The set of remembered associations may be a fixed-size set controlled by a replacement algorithm or a fixed set, depending on the nature of the function and its use. A function can only be memoized if it is referentially transparent; that is, only if calling the function has exactly the same effect as replacing that function call with its return value. (Special case exceptions to this restriction exist, however.) While related to lookup tables, since memoization often uses such tables in its implementation, memoization populates its cache of results transparently on the fly, as needed, rather than in advance.

Memoization is a way to lower a function's *time* cost in exchange for *space* cost; that is, memoized functions become optimized for *speed* in exchange for a higher use of computer memory *space*. The time/space "cost" of algorithms has a specific name in computing: *computational complexity*. All functions have a computational complexity in *time* (i.e. they take time to execute) and in *space*.

Although a time-space trade-off occurs (i.e., space used is speed gained), this differs from some other optimizations that involve time-space trade-off, such as strength reduction, in that memoization is a run-time rather than compile-time optimization. Moreover, strength reduction potentially replaces a costly operation such as multiplication with a less costly operation such as addition, and the results in savings can be highly machine-dependent, non-portable across machines, whereas memoization is a more machine-independent, cross-platform strategy.

Consider the following pseudocode function to calculate the factorial of *n*:

```
function factorial (n is a non-negative integer)
    if n is 0 then
        return 1 [by the convention that 0! = 1]
    else
        return factorial(n – 1) times n [recursively invoke factorial
                                         with the parameter 1 less than n]
    end if
end function
```

For every integer *n* such that `n≥0`, the final result of the function `factorial` is invariant; if invoked as `x = factorial(3)`, the result is such that *x* will *always* be assigned the value 6. A non-memoized version of the above, given the nature of the recursive algorithm involved, would require *n + 1* invocations of `factorial` to arrive at a result, and each of these invocations, in turn, has an associated cost in the time it takes the function to return the value computed. Depending on the machine, this cost might be the sum of:

1. The cost to set up the functional call stack frame.
2. The cost to compare *n* to 0.
3. The cost to subtract 1 from *n*.
4. The cost to set up the recursive call stack frame. (As above.)
5. The cost to multiply the result of the recursive call to `factorial` by *n*.
6. The cost to store the return result so that it may be used by the calling context.

In a non-memoized implementation, *every* top-level call to `factorial` includes the cumulative cost of steps 2 through 6 proportional to the initial value of *n*.

A memoized version of the `factorial` function follows:

```
function factorial (n is a non-negative integer)
    if n is 0 then
        return 1 [by the convention that 0! = 1]
    else if n is in lookup-table then
        return lookup-table-value-for-n
    else
        let x = factorial(n – 1) times n [recursively invoke factorial
                                          with the parameter 1 less than n]
        store x in lookup-table in the nth slot [remember the result of n! for later]
        return x
    end if
end function
```

In this particular example, if `factorial` is first invoked with 5, and then invoked later with any value less than or equal to five, those return values will also have been memoized, since `factorial` will have been called recursively with the values 5, 4, 3, 2, 1, and 0, and the return values for *each* of those will have been stored. If it is then called with a number greater than 5, such as 7, only 2 recursive calls will be made (7 and 6), and the value for 5! will have been stored from the previous call. In this way, memoization allows a function to become more time-efficient the more often it is called, thus resulting in eventual overall **speed up**.

Some other considerations

Functional programming

Memoization is heavily used in compilers for functional programming languages, which often use call by name evaluation strategy. To avoid overhead with calculating argument values, compilers for these languages heavily use auxiliary functions called thunks to compute the argument values, and memoize these functions to avoid repeated calculations.

Automatic memoization

While memoization may be added to functions *internally* and *explicitly* by a computer programmer in much the same way the above memoized version of `factorial` is implemented, referentially transparent functions may also be automatically memoized *externally*.[1] The techniques employed by Peter Norvig have application not only in Common Lisp (the language in which his paper demonstrated automatic memoization), but also in various other programming languages. Applications of automatic memoization have also been formally explored in the study of term rewriting[6] and artificial intelligence.[7]

In programming languages where functions are first-class objects (such as Lua, Python, or Perl [1] (http://perl.plover.com/MiniMemoize/memoize.html)), automatic memoization can be implemented by replacing (at run-time) a function with its calculated value once a value has been calculated for a given set of parameters. The function that does this value-for-function-object replacement can generically wrap any referentially transparent function. Consider the following pseudocode (where it is assumed that functions are first-class values):

```
function memoized-call (F is a function object parameter)
   if F has no attached array values then
      allocate an associative array called values;
      attach values to F;
   end if;

   if F.values[arguments] is empty then
      F.values[arguments] = F(arguments);
   end if;

   return F.values[arguments];
end function
```

In order to call an automatically memoized version of `factorial` using the above strategy, rather than calling `factorial` directly, code invokes `memoized-call(factorial(n))`. Each such call first checks to see if a holder array has been allocated to store results, and if not, attaches that array. If no entry exists at the position `values[arguments]` (where `arguments` are used as the key of the associative array), a *real* call is made to `factorial` with the supplied arguments. Finally, the entry in the array at the key position is returned to the caller.

The above strategy requires *explicit* wrapping at each call to a function that is to be memoized. In those languages that allow closures, memoization can be effected *implicitly* by a functor factory that returns a wrapped memoized function object. In pseudocode, this can be expressed as follows:

```
function construct-memoized-functor (F is a function object parameter)
   allocate a function object called memoized-version;

   let memoized-version(arguments) be
      if self has no attached array values then [self is a reference to this object]
         allocate an associative array called values;
         attach values to self;
      end if;

      if self.values[arguments] is empty then
         self.values[arguments] = F(arguments);
      end if;

      return self.values[arguments];
   end let;

   return memoized-version;
end function
```

Rather than call `factorial`, a new function object `memfact` is created as follows:

```
memfact = construct-memoized-functor(factorial)
```

The above example assumes that the function `factorial` has already been defined *before* the call to `construct-memoized-functor` is made. From this point forward, `memfact(n)` is called whenever the factorial of *n* is desired. In languages such as Lua, more sophisticated techniques exist which allow a function to be replaced by a new function with the same name, which would permit:

```
factorial = construct-memoized-functor(factorial)
```

Essentially, such techniques involve attaching the *original function object* to the created functor and forwarding calls to the original function being memoized via an alias when a call to the actual function is required (to avoid endless recursion), as illustrated below:

```
function construct-memoized-functor (F is a function object parameter)
    allocate a function object called memoized-version;

    let memoized-version(arguments) be
       if self has no attached array values then [self is a reference to this object]
          allocate an associative array called values;
          attach values to self;
          allocate a new function object called alias;
          attach alias to self; [for later ability to invoke F indirectly]
          self.alias = F;
       end if;

       if self.values[arguments] is empty then
          self.values[arguments] = self.alias(arguments); [not a direct call to F]
       end if;

       return self.values[arguments];
    end let;

    return memoized-version;
end function
```

(Note: Some of the steps shown above may be implicitly managed by the implementation language and are provided for illustration.)

Parsers

When a top-down parser tries to parse an ambiguous input with respect to an ambiguous context-free grammar (CFG), it may need an exponential number of steps (with respect to the length of the input) to try all alternatives of the CFG in order to produce all possible parse trees. This eventually would require exponential memory space. Memoization was explored as a parsing strategy in 1991 by Norvig, who demonstrated that an algorithm similar to the use of dynamic programming and state-sets in Earley's algorithm (1970), and tables in the CYK algorithm of Cocke, Younger and Kasami, could be generated by introducing automatic memoization to a simple backtracking recursive descent parser to solve the problem of exponential time complexity.[1] The basic idea in Norvig's approach is that when a parser is applied to the input, the result is stored in a memotable for subsequent reuse if the same parser is ever reapplied to the same input. Richard Frost also used memoization to reduce the exponential time complexity of parser combinators, which can be viewed as "Purely Functional Top-Down Backtracking" parsing technique.[8] He showed that basic memoized parser combinators can be used as building blocks to construct complex parsers as executable specifications of CFGs.[9][10] It was again explored in the context of parsing in 1995 by Johnson and Dörre.[11][12] In 2002, it was examined in considerable depth by Ford in the form called packrat parsing.[13]

In 2007, Frost, Hafiz and Callaghan[2] described a top-down parsing algorithm that uses memoization for refraining redundant computations to accommodate any form of ambiguous CFG in polynomial time ($\Theta(n^4)$ for left-recursive grammars and $\Theta(n^3)$ for non left-recursive grammars). Their top-down parsing algorithm also requires polynomial space for potentially exponential ambiguous parse trees by 'compact representation' and 'local ambiguities grouping'. Their compact representation is comparable with Tomita's compact representation of bottom-up parsing.[14] Their use of memoization is not only limited to retrieving the previously computed results when a parser is applied to a same input position repeatedly (which is essential for polynomial time requirement); it is specialized to perform the following additional tasks:

- The memoization process (which could be viewed as a 'wrapper' around any parser execution) accommodates an ever-growing **direct left-recursive** parse by imposing depth restrictions with respect to input length and current input position.
- The algorithm's memo-table 'lookup' procedure also determines the reusability of a saved result by comparing the saved result's computational context with the parser's current context. This contextual comparison is the key to accommodate **indirect (or hidden) left-recursion**.
- When performing a successful lookup in a memotable, instead of returning the complete result-set, the process only returns the references of the actual result and eventually speeds up the overall computation.
- During updating the memotable, the memoization process groups the (potentially exponential) ambiguous results and ensures the polynomial space requirement.

Frost, Hafiz and Callaghan also described the implementation of the algorithm in PADL'08[3] as a set of higher-order functions (called parser combinators) in Haskell, which enables the construction of directly executable specifications of CFGs as language processors. The importance of their polynomial algorithm's power to accommodate 'any form of ambiguous CFG' with top-down parsing is vital with respect to the syntax and semantics analysis during natural language processing. The X-SAIGA (http://www.cs.uwindsor.ca/~hafiz/proHome.html) site has more about the algorithm and implementation details.

While Norvig increased the *power* of the parser through memoization, the augmented parser was still as time complex as Earley's algorithm, which demonstrates a case of the use of memoization for something other than speed optimization. Johnson and Dörre[12] demonstrate another such non-speed related application of memoization: the use of memoization to delay linguistic constraint resolution to a point in a parse where sufficient information has been accumulated to resolve those constraints. By contrast, in the speed optimization application of memoization, Ford demonstrated that memoization could guarantee that parsing expression grammars could parse in linear time even those languages that resulted in worst-case backtracking behavior.[13]

Consider the following grammar:

```
S → (A c) | (B d)
A → X (a|b)
B → X b
X → x [X]
```

(Notation note: In the above example, the production S → (A **c**) | (B **d**) reads: "An *S* is either an *A* followed by a **c** or a *B* followed by a **d**." The production X → **x** [X] reads "An *X* is an **x** followed by an optional *X*.")

This grammar generates one of the following three variations of string: *xac*, *xbc*, or *xbd* (where *x* here is understood to mean *one or more* x*'s*.) Next, consider how this grammar, used as a parse specification, might effect a top-down, left-right parse of the string *xxxxxbd*:

> The rule *A* will recognize *xxxxxb* (by first descending into *X* to recognize one *x*, and again descending into *X* until all the *x*'s are consumed, and then recognizing the *b*), and then return to *S*, and fail to recognize a *c*. The next clause of *S* will then descend into B, which in turn **again descends into *X*** and recognizes the *x*'s by means of many recursive calls to *X*, and then a *b*, and returns to *S* and finally recognizes a *d*.

The key concept here is inherent in the phrase **again descends into *X***. The process of looking forward, failing, backing up, and then retrying the next alternative is known in parsing as backtracking, and it is primarily backtracking that presents opportunities for memoization in parsing. Consider a function `RuleAcceptsSomeInput(Rule, Position, Input)`, where the parameters are as follows:

- `Rule` is the name of the rule under consideration.
- `Position` is the offset currently under consideration in the input.
- `Input` is the input under consideration.

Let the return value of the function `RuleAcceptsSomeInput` be the length of the input accepted by `Rule`, or 0 if that rule does not accept any input at that offset in the string. In a backtracking scenario with such memoization, the parsing process is as follows:

> When the rule *A* descends into *X* at offset 0, it memoizes the length 5 against that position and the rule *X*. After having failed at *d*, *B* then, rather than descending again into *X*, queries the position 0 against rule *X* in the memoization engine, and is returned a length of 5, thus saving having to actually descend again into *X*, and carries on *as if* it had descended into *X* as many times as before.

In the above example, one or *many* descents into *X* may occur, allowing for strings such as *xxxxxxxxxxxxxxxxbd*. In fact, there may be *any number* of *x*'s before the *b*. While the call to S must recursively descend into X as many times as there are *x*'s, *B* will never have to descend into X at all, since the return value of `RuleAcceptsSomeInput(X, 0, xxxxxxxxxxxxxxxxbd)` will be 16 (in this particular case).

Those parsers that make use of syntactic predicates are also able to memoize the results of predicate parses, as well, thereby reducing such constructions as:

```
S → (A)? A
A → /* some rule */
```

to one descent into *A*.

If a parser builds a parse tree during a parse, it must memoize not only the *length* of the input that matches at some offset against a given rule, but also must store the sub-tree that is generated by that rule at that offset in the input, since subsequent calls to the rule by the parser will not actually descend and rebuild that tree. For the same reason, memoized parser algorithms that generate calls to external code (sometimes called a semantic action routine) when a rule matches must use some scheme to ensure that such rules are invoked in a predictable order.

Since, for any given backtracking or syntactic predicate capable parser not every grammar will *need* backtracking or predicate checks, the overhead of storing each rule's parse results against every offset in the input (and storing the parse tree if the parsing process does that implicitly) may actually *slow down* a parser. This effect can be mitigated by explicit selection of those rules the parser will memoize.[15]

See also

- Computational complexity theory – more information on algorithm complexity
- Approximate computing – category of techniques to improve efficiency
- Strength reduction – a compiler optimization that replaces a costly operation with an equivalent, less costly one
- Partial evaluation – a related technique for automatic program optimization
- Lazy evaluation – shares some concepts with memoization
- Lookup table – a key data structure used in memoization
- Flyweight pattern – an object programming design pattern, that also uses a kind of memoization
- Director string – rapidly locating free variables in expressions
- Dynamic programming – some applications of memoizing techniques
- Hashlife – a memoizing technique to speed up the computation of cellular automata
- Higher-Order Perl – a free book by Mark Jason Dominus contains an entire chapter on implementing memoization, along with some background
- Materialized view – analogous caching in database queries

References

1. Norvig, Peter, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, Vol. 17 No. 1, pp. 91–98,

March 1991.
2. Frost, Richard, Hafiz, Rahmatullah, and Callaghan, Paul. " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109 – 120, June 2007, Prague.
3. Frost, Richard, Hafiz, Rahmatullah, and Callaghan, Paul. " Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902/2008, Pages: 167–181, January 2008, San Francisco.
4. Warren, David. "Tabling and Datalog Programming (http://www.cs.sunysb.edu/~warren/xsbbook/node14.html)". Accessed 29 May 2009.
5. Michie, Donald, "Memo Functions and Machine Learning," *Nature*, No. 218, pp. 19–22, 1968.
6. Hoffman, Berthold, "Term Rewriting with Sharing and Memoïzation," *Algebraic and Logic Programming: Third International Conference, Proceedings*, H. Kirchner and G. Levi (eds.), pp. 128–142, Volterra, Italy, 2–4 September 1992.
7. Mayfield, James, *et al.*, *Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems*, Proceedings of the Eleventh IEEE Conference on Artificial Intelligence for Applications (CAIA '95), pp. 87-93, 1995.
8. Frost, Richard. and Szydlowski, Barbara. "Memoizing Purely Functional Top-Down Backtracking Language Processors. " "Sci. Comput. Program. " 1996 – 27(3): 263–288.
9. Frost, Richard. "Using Memoization to Achieve Polynomial Complexity of Purely Functional Executable Specifications of Non-Deterministic Top-Down Parsers. " "SIGPLAN Notices" 29(4): 23–30.
10. Frost, Richard. "Monadic Memoization towards Correctness-Preserving Reduction of Search. " "Canadian Conference on AI 2003." p 66-80.
11. Johnson, Mark, "Memoization of Top-Down Parsing," *Computational Linguistics*, Vol. 21 No. 3, pp. 405–417, September 1995.
12. Johnson, Mark & Dörre, Jochen, "Memoization of Coroutined Constraints," *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, Cambridge, Massachusetts, 1995.
13. Ford, Bryan, *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Master's thesis, Massachusetts Institute of Technology, September, 2002.
14. Tomita, Masaru. "Efficient Parsing for Natural Language." *Kluwer, Boston, MA*, 1985.
15. Acar, Umut A. A. *et al.*, "Selective Memoization," *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, Louisiana, pp. 14–25, 15–17 January 2003.

External links

Examples of memoization in various programming languages

- groovy.lang.Closure#memoize() (http://groovy.codehaus.org/api/groovy/lang/Closure.html#memoize()) – Memoize is a Groovy 1.8 language feature.
- Memoize (http://www.cliki.net/memoize) – Memoize is a small library, written by Tim Bradshaw, for performing memoization in Common Lisp.
- IncPy (http://www.pgbovine.net/incpy.html) – A custom Python interpreter that performs automatic memoization (with no required user annotations)
- Dave Herman's Macros for defining memoized procedures (http://planet.racket-lang.org/display.ss?package=memoize.plt&owner=dherman) in Racket.
- Memoize.pm (https://metacpan.org/module/Memoize) – a Perl module that implements memoized functions.
- Java memoization (http://www.onjava.com/pub/a/onjava/2003/08/20/memoization.html) – an example in Java using dynamic proxy classes to create a generic memoization pattern.
- memoization.java (https://github.com/sebhoss/memoization.java) - A Java memoization library.
- C++Memo (http://projects.giacomodrago.com/c++memo/) – A C++ memoization framework.
- C-Memo (http://sourceforge.net/projects/c-memo/) – Generic memoization library for C, implemented using pre-processor function wrapper macros.
- Tek271 Memoizer (http://www.tek271.com/software/java/memoizer) – Open source Java memoizer using annotations and pluggable cache

implementations.
- memoizable (https://github.com/dkubb/memoizable) – A Ruby gem that implements memoized methods.
- Python memoization (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52201) – A Python example of memoization.
- OCaml memoization (http://martin.jambon.free.fr/309/pa_memo.ml.html) – Implemented as a Camlp4 syntax extension.
- Memoization in Lua (http://lua-users.org/wiki/FuncTables) – Two example implementations of a general memoize function in Lua.
- Memoization in Mathematica (http://szhorvat.net/pelican/memoization-in-mathematica.html) – Memoization and limited memoization in Mathematica.
- Memoization in Javascript (http://archive.is/FUbks) – Extending the Function prototype in JavaScript (archived version of http://talideon.com/weblog/2005/07/javascript-memoization.cfm).
- Memoization in Javascript (http://odhyan.com/blog/2010/09/caching-in-javascript-yui-cache/) – Examples of memoization in javascript using own caching mechanism and using the YUI library
- X-SAIGA (http://www.cs.uwindsor.ca/~hafiz/proHome.html) – eXecutable SpecificAtIons of GrAmmars. Contains publications related to top-down parsing algorithm that supports left-recursion and ambiguity in polynomial time and space.
- Memoization in Scheme (http://codeimmersion.i3ci.hampshire.edu/2009/10/09/memoization/) – A Scheme example of memoization on a class webpage.
- Memoization in Combinatory Logic (http://code.google.com/p/clache/) – A web service to reduce Combinatory Logic while memoizing every step in a database.
- MbCache (http://code.google.com/p/mbcache/) – Cache method results in .NET.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Memoization&oldid=755028395"

Categories: Software optimization | Computer performance